Project 3

Syeda Ali
Branimir Maximov
Kevin Calva

Part A) ISA intro

1. Introduction.

Name of Architecture: SBK which is the first letter of all of our names; Syeda, Branimir, Kevin

Overall Philosophy: Complete ISA with a python simulator to execute the 2 programs

Specific goals strived for and achieved:

Modular Exponentiation

Best match score and count

2. Instruction list. Give all the instructions, their formats, opcodes, and an example.

Instr	**Format**	**Op**	**Example**	
ADD	100 xx, yy Rx Ry	100	ADD r3, r2	r3 = r3 + r2
LWD	001 xx, yy Rx Ry	001	LWD r0,[r1]	r0 = M[r1]
SWD	011 xx, yy Rx Ry	011	SWD r1,[r2]	M[r2] = r1
INIT	101 xx, yy Rx Ry	101	INIT r0, 1	r0 = 1
ADDI	111 xx, yy Rx Ry	111	ADDI r0, 2	r0 = r0 + 2
SLE	110 xx, yy Rx Ry	110	SLE r1, r2	If r1 < r2 r3 = 1 Else r3 = 0
JIF	010 xxxx imm	010	JIF -7	If r3 = 1 => PC = PC +imm Else PC = PC + 1
XOR	0001 x, yy Rx Ry	0001	XOR r1, r2	r1 ^= r2
SLER	0000 xx, y Rx Ry (Ry cannot be r1) (Rx must be r0)	0000	SLER r1, r0	If r1 < r0 r3 = 1 Else r3 = 0

SUBR0	00000 yy Ry	00000	SUBR0 r2	r0 = r0 - r2
ADDN	1111100	1111100	ADDN	r3 = r3 - 1
CNTR0	0001000	0001000	CNTR0	Count the number of bits with a value of ‘1’ in r0
HLT	0001111	0001111	HLT	Stop the program

3. Register design. How many registers are supported? Is there anything special about the registers?

There are 4 registers supported. The registers are mostly multi-purpose except r3 is used as the register to store the result of SLE and SLER. r3 is also the register that is checked to determine if a jump occurs or not (JIF). Some registers cannot be used for certain instructions or are the only register that can be used as one of the operands.

4. Control flow (branches). What types of branches are supported? How are the target addresses calculated? What is the maximum branch distance supported? Give examples of your assembly branch instructions and their corresponding machine code.

The only branching instruction is JIF. This checks if the value in r3 is 1, then branches if so. If the value is anything else, then nothing is done. There is no additional calculation for this instruction, an immediate is sent to an adder that sums the immediate with PC. This supports a jump of up to 7 or -7.

- Example: JIF 5
- Machine Code: 0100101
- This example jumps 5 instructions ahead if r3 = 1.

5. Data memory addressing modes. What addressing modes are supported for data memory? How are the addresses calculated? Give examples of your assembly load / store instructions and their corresponding machine code.

Load word and store word are supported for data memory. There is no additional calculation, a register is used to load to memory.

- Example: LWD r2, [r3]

- Machine Code: 0011011
- This loads the contents of register r2 into the memory location corresponding to the value in r3.
- Example: SWD r0, [r1]
- Machine Code: 0110001
- This stores the contents of r0 into the memory location corresponding to the value in r1.

Part B) Answers to questions

1. What are the most significant advantages of your ISA (with regard to the two programs, hardware implementation, ease of programming, etc)? What are the main limitations? What are the main compromises that you have done to make things work, rather than perfecting everything?

The most significant advantages to the ISA are that it uses only a few instructions and registers which would result in more concise and organized code. The limitations are that only seven bits can be used for machine code instruction line and these seven bits need to be manipulated to execute two complete programs. Additionally three instructions in the ISA design were hard coded which allowed for fewer options of bit combinations when building other instructions.

2. What have you done towards the goals of low DIC and HW simplification? What could have been done differently to better optimize for each of the two goals, if to start over?

We optimized for the two goals by using a limited of registers so that the code would be more concise and organized in addition to using parity bits which helped reduce significant technical errors.

3. Reflect on this project(1-3) experience:

- What did you learn from this project? What was the best / worst thing about it?
    i. How memory allocation and deallocation is used in Python. This was very new information which meant that we gained a lot of exposure to
- What advice would you give to someone taking this project in a future semester?
    I. Start early
- How would you describe the value of this project experience in a job interview?
    I. It teaches valuable skills and gets you thinking about hardware more

Part C) Software package:

1. Algorithms (in assembly code) of the two programs. Make sure your assembly format is either obvious or well described, and that the code is well commented.

Extra credits: provide a convincing estimation:

i. on the dynamic instruction count for P1 (ME) with P = ~1000 and Q = ~500

ii. on the worst-case scenario of dynamic instruction count for P2(BMC).

2. Machine code for each of the programs. We will not correct/grade the machine code. You will also not be able to verify whether your code works correctly or not in this project (without a simulator). Therefore, you have to rely on the help of the disassembler, strive to make your algorithm simple and easy to understand, as well as pursue the sw-hw "codesign" – this will avoid putting tremendous complexity at either the software or the hardware end.

Algorithm and Machine Code for Program 1:

#Assume everything is equal to zero at first
#$t0 = 00
#$t1 = 01
#$t2 = 10
#$t3 = 11

addi $t0, 1	0 100 00 01
addi $t1, 1	0 100 01 01
loadi 0(0x2000)	0 011 0000
addi $t2, 3	0 100 10 11
addi $t2, 2	0 100 10 10
addi $t3, 3	0 100 11 11
addi $t3, 3	0 100 11 11
addi $t3, 3	0 100 11 11
addi $t3, 3	0 100 11 11
addi $t3, 3	0 100 11 11
addi $t3, 2	0 100 11 10
loop:	
bne $s0	1 1110000
BezDec 7	0 0100 111
next:	
bne $t2	1 110 11 10
BezDec 4	0 0100 100
add $t1, $t0	1 001 01 00
subi $t2	0 101 10 10
jump '*next*'	0 010 1000
next2:	
sub $t3, $t1	1 101 11 01
bne $s0	1 1110000

BezDec 7	0 0100 111
slt $t3	1 100 01 11
add $t3, $t1	1 001 11 01
BezDec 3	0 0100 011
sub $t1, $t3	1 101 01 11
jump ‘next2’	0 010 0111

down:

addi $t2, 3	0 100 10 11
addi $t2, 2	0 100 10 10
bne $s0	1 1110000
BezDec 5	0 0100 101
subIn $s0, 1	1 0110110
sub $t0, $t0	1 101 00 00
add $t0, $t1	1 001 00 01
jump *‘loop’*	0 1010101

exit:

store $t1, 0(0x2004)	1 000 0100
halt	0 000 0000

Algorithm and Machine Code for Program 2:

#Assume everything is equal to zero at first
#$t0 = 00
#$t1 = 01
#$t2 = 10
#$t3 = 11

addi $t0, 3	0 100 00 11
addi $t0, 3	0 100 00 11
addi $t0, 3	0 100 00 11
addi $t0, 3	0 100 00 11
addi $t0, 3	0 100 00 11
addi $t0, 3	0 100 00 11
addi $t0, 2	0 100 00 10
addi $t1, 3	0 100 01 11
addi $t1, 3	0 100 01 11
addi $t1, 3	0 100 01 11
addi $t1, 3	0 100 01 11
addi $t1, 3	0 100 01 11
addi $t1, 3	0 100 01 11
addi $t1, 3	0 100 01 11

addi $t1, 3	0 100 01 11
addi $t1, 3	0 100 01 11
addi $t1, 3	0 100 01 11
addi $t1, 2	0 100 01 10
next:	
loadi 0(0x200C)	0 011 1100
next2:	
loadi 0(0x2000)	0 011 0000
load $t2	1 01110 10
subi $t1	0 101 10 01
xor $t2, $t3	0 110 10 11
sub $t0, $0	1 101 00 00
bne $t0	1 110 11 00
BezDec 7	0 0100 111
next3:	
bne $t1	1 110 11 01
BezDec 5	0 0100 101
subi $t1	0 101 10 01
andi5 1	1 11110 01
srl5 1	1 01111 01
jump ‘next3’	1 010 0101
next4:	
bne $t0	1 110 11 00
BezDec 7	0 0100 111
loadi 0(0x2010)	0 011 0000
load $t3	1 01110 11
subi $t0	0 101 10 00
loadi 0(0x2004)	0 011 0100
blt4 $t1	0 00001 01
bne $t1	1 110 11 01
bne $t0	1 110 11 00
BezDec 7	0 0100 111
BezDec 5	0 0100 101
beq4 $t1	1 1000 01
bne $t1	1 110 11 01
BezDec 7	0 0100 111
jump ‘first branch’	0 1010101
score:	

sub $t2, $t2	1 101 10 10
bne $t0	1 110 11 00
BezDec 7	0 0100 111
addi $t2, 1	0 100 01 01
store $t1, 0(0x2004)	1 000 0100
bne $t1	1 110 11 01
BezDec 3	0 0100 011
store $t2, 0(0x2008)	1 000 1000
jump ‘first branch’	0 1010101

best count:

sub $t2, $t2	1 101 10 10
addi $t2, 1	0 100 01 01
bne $t0	1 110 11 00
BezDec 3	0 0100 011
store $t2, 0(0x2008)	1 000 1000
jump ‘first branch’	0 1010101

exit:

halt	0 000 0000

3. Output of your Python disassembler for each program. This should be a line-by-line explanation of the machine code, what is done by each line of code.

P1

```
ECE 366 Group 8
 2 = disassembler
Please enter the mode of Program: 2
Mode selected:  ECE 366 Group 8 Disassembler
----------------
Machine code: 01000001

addi R0, 1
----------------
Machine code: 01000101

addi R1, 1
----------------
Machine code: 00110000

load 0
----------------
Machine code: 01001011

addi R2, 3
----------------
Machine code: 01001010

addi R2, 2
----------------
Machine code: 01001111
```

addi R3, 3

Machine code: 01001111

addi R3, 3

Machine code: 01001111

addi R3, 3

Machine code: 01001111

addi R3, 3

Machine code: 01001111

addi R3, 3

Machine code: 01001110

addi R3, 2

Machine code: 11110000

bne $R4

Machine code: 00100111

bezDec 7

Machine code: 11101110

bne R3

Machine code: 00100100

bezDec 4

Machine code: 10010100

add R1, R0

Machine code: 01011010

subi R2

Machine code: 10101000

jump1,8

Machine code: 11011101

sub R3, R1

Machine code: 11110000

bne $R4

Machine code: 00100111

bezDec 7

Machine code: 11000111

slt R3

Machine code: 10011101

add R3, R1

Machine code: 00100011

bezDec 3

Machine code: 11010111

sub R1, R3

Machine code: 10100111

jump1,7

Machine code: 01001011

addi R2, 3

Machine code: 01001010

addi R2, 2

Machine code: 11110000

bne $R4

Machine code: 00100101

bezDec 5

Machine code: 10110110

subln $R4

Machine code: 11010000

sub R0, R0

Machine code: 00010001

srl R1

Machine code: 01010101

jump 'first branch'

Machine code: 10000100

store R1, R0

Machine code: 00000000

halt
---
disassembler is being done

P2 output

ECE 366 Group 8
 2 = disassembler
Please enter the mode of Program: 2

Mode selected: Please enter the which program 1 or 2: 2
Mode selected: ECE 366 Group 8 Disassembler

Machine code: 01000011

addi R0, 3

Machine code: 01000011

addi R0, 3

Machine code: 01000011

addi R0, 3

Machine code: 01000011

addi R0, 3

Machine code: 01000011

addi R0, 3

Machine code: 01000011

addi R0, 3

Machine code: 01000010

addi R0, 2

Machine code: 01000111

addi R1, 3

Machine code: 01000111

addi R1, 3

Machine code: 01000111

addi R1, 3

Machine code: 01000111

addi R1, 3

Machine code: 01000111

addi R1, 3

Machine code: 01000111

addi R1, 3

Machine code: 01000111

addi R1, 3

Machine code: 01000111

addi R1, 3

Machine code: 01000111

addi R1, 3

Machine code: 01000111

addi R1, 3

Machine code: 01000110

addi R1, 2

Machine code: 00111100

loadi 12

Machine code: 00110000

loadi 0

Machine code: 10111010

load R2

Machine code: 01011001

subi R1

Machine code: 01101011

xor R2,R3

Machine code: 11010000

sub R0, R0

Machine code: 11101100

bne R0

Machine code: 00100111

bezDec 7

Machine code: 11101101

bne R0

Machine code: 00100101

bezDec 5

Machine code: 01011001

subi R1

Machine code: 01111001

xori R2, R1

Machine code: 10111101

srl5 1

Machine code: 10100101

jump 5

Machine code: 11101100

bne R2

Machine code: 00100111

bezDec 7

Machine code: 00110000

loadi 0

Machine code: 10111011

load R3

Machine code: 01011000

subi R0

Machine code: 00110100

loadi 4

Machine code: 00000101

blt4 R1

Machine code: 11101101

bne R1

Machine code: 11101100

bne R1

Machine code: 00100111

bezDec 7

Machine code: 00100101

bezDec 5

Machine code: 11000001

beq R1

Machine code: 11101101

bne R1

Machine code: 00100111

bezDec 7

Machine code: 11010101

sub R1, R1

Machine code: 11011010

sub R2, R2

Machine code: 11101100

bne R2

Machine code: 00100111

bezDec 7

Machine code: 01000101

addi R1, 1

Machine code: 10000100

store R1, R0

Machine code: 11101101

bne R1

Machine code: 00100011

bezDec 3

Machine code: 10001000

store R2, R0

Machine code: 11010101

sub R1, R1

Machine code: 11011010

```
sub R2, R2
----------
Machine code: 01000101

addi R1, 1
----------
Machine code: 11101100

bne R1
----------
Machine code: 00100011

bezDec 3
----------
Machine code: 10001000

store R2, R0
----------
Machine code: 11010101

sub R1, R1
----------
Machine code: 00000000
halt
----------
disassembler is being done
```

4. Python code for your ISA’s disassembler.

ISA.py

```
#ISA design for AVS

def disassembler(M,Nlines):
        print("ECE 366 Group 8 Disassembler")
        print("----------")
   #print the instructions
        Rx = 0
        Ry= 0
        imm = 0
        for i in range(Nlines):

        fetch =M[i]
        print("Machine code: " + M[i])
        if(fetch[0:4] == "0011"): # load imm
        imm= int(fetch[4:8],2)
        print("loadi "+ str(imm))

        elif(fetch[0:4] == "1000"):#store Rx, Ry
        Rx= int(fetch[4:6],2)
        Ry= int(fetch[6:8],2)
        print("store R" + str(Rx)+ ", R" + str(Ry))

        elif(fetch[0:6] == "000100"):#srl Rx
        Rx= int(fetch[6:8],2)
        print("srl R" + str(Rx))
```

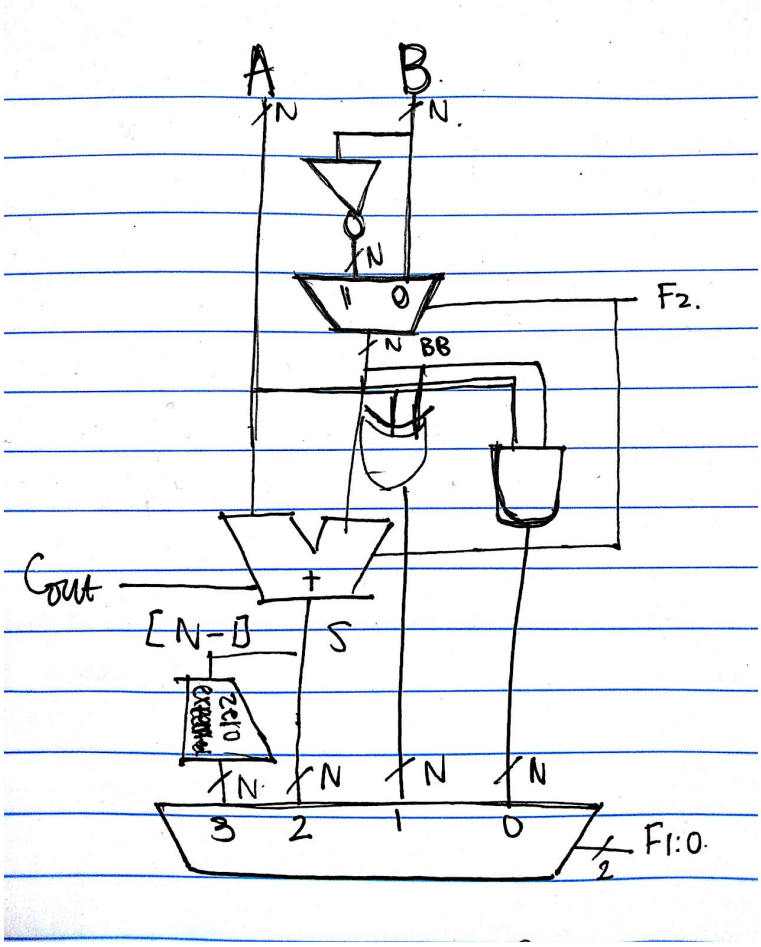
```
elif(fetch[0:4] == "0100"):#addi Rx, imm
Rx= int(fetch[4:6],2)
imm= int(fetch[6:8],2)
print("addi R" + str(Rx)+ ", " + str(imm))

elif(fetch[0:6] == "010110"):#subi Rx
Ry= int(fetch[6:8],2)
print("subi R" + str(Ry))

elif(fetch[0:6] == "111011"):#bne Rx
print("bne R" + str(Rx))

elif(fetch[0:6] == "110001"):#slt Rx
Rx= int(fetch[6:8],2)
print("slt R" + str(Rx))

elif(fetch[0:5] == "00100"):#BezDec imm
imm= int(fetch[5:8],2)
print("bezDec " + str(imm))

elif(fetch[0:4] == "0111"):#xori Rx, imm
Rx= int(fetch[4:6],2)
imm= int(fetch[6:8],2)
print("xori R" + str(Rx)+ ", R" + str(imm))

elif(fetch[0:4] == "0111"):#andi Rx, imm
Rx= int(fetch[4:6],2)
imm= int(fetch[6:8],2)
print("andi" + str(Rx)+ "," + str(imm))

elif(fetch[0:4] == "1010"):#jump 'branch'(imm)
imm= int(fetch[4:8],2)
print("jump " + str(imm))

elif(fetch[0:4] == "1001"):#add RX, Ry
Rx= int(fetch[4:6],2)
Ry= int(fetch[6:8],2)
print("add R" + str(Rx)+ ", R" + str(Ry))

elif(fetch[0:4] == "1101"):#sub Rx, Ry
Rx= int(fetch[4:6],2)
Ry= int(fetch[6:8],2)
print("sub R" + str(Rx)+ ", R" + str(Ry))

elif(fetch[0:8] == "10110110"):#subln R4

print("subln $R4")

elif(fetch[0:8] == "11110000"):#bne R4
print("bne $R4" )

elif(fetch[0:8] == "01010101"):# jump ' first branch'
print("jump 'first branch' ")

elif(fetch[0:8] == "00000000"):#halt
Rx= int(fetch[4:6],2)
Ry= int(fetch[6:8],2)
```

```
            print("halt")

            elif(fetch[0:6]=="101110"): #load Rx
            Rx= int(fetch[6:8],2)
            print("load R" + str(Rx))

            elif(fetch[0:6]=="000001"): #blt4 Rx
            Rx= int(fetch[6:8],2)
            print("blt4 R" + str(Rx))

            elif(fetch[0:6] == "110000"): #beq4 Rx
            Rx = int(fetch[6:8],2)
            print("beq R" + str(Rx))

            elif(fetch[0:6] == "111110"): #andi5 imm
            imm= int(fetch[6:8],2)
            print("and5i R" + str(imm))

            elif(fetch[0:6] == "101111"): #srl5 imm
            imm=int(fetch[6:8])
            print("srl5 " + str(imm))

            elif(fetch[0:4] == "0110"): #xor Rx, Ry
            Rx =int(fetch[4:6],2)
            Ry =int(fetch[6:8],2)
            print("xor R" + str(Rx) + ",R" + str(Ry))

            else:
            print("Instruction set not supported")
            print("----------------")


#def assembler(I,Nlines):
#def simulator(I,Nsteps,debug_mode,Memory):
def main():
            instr_file = open("P1_Instruction.txt","r")
            data_file = open("project2_group8_p1_bin.txt" ,"r")
            data_file2 = open("project2_group8_p2_bin.txt", "r")
            #we need a file for the data set
            #Nsteps = 3  #How many cycles to run before output
            Nlines = 0   #How may instrs total in input.txt
            Instructions = [] #all instructions will be stored here
            Memory = []
            print( " ECE 366 Group 8")
   #print( " 1 = simulator")
            print( " 2 = disassembler")
            #print( " 3 = assembler")

            mode= int(input( "Please enter the mode of Program: "))
            print( "Mode selected: ",end=" ")
            modedis= int(input( "Please enter the which program 1 or 2:  "))
            print( "Mode selected: ", end=" ")

            #if(mode== 1):
            #print("Simulator")
            #elif(mode== 2):
            # print( "disassembler")
```

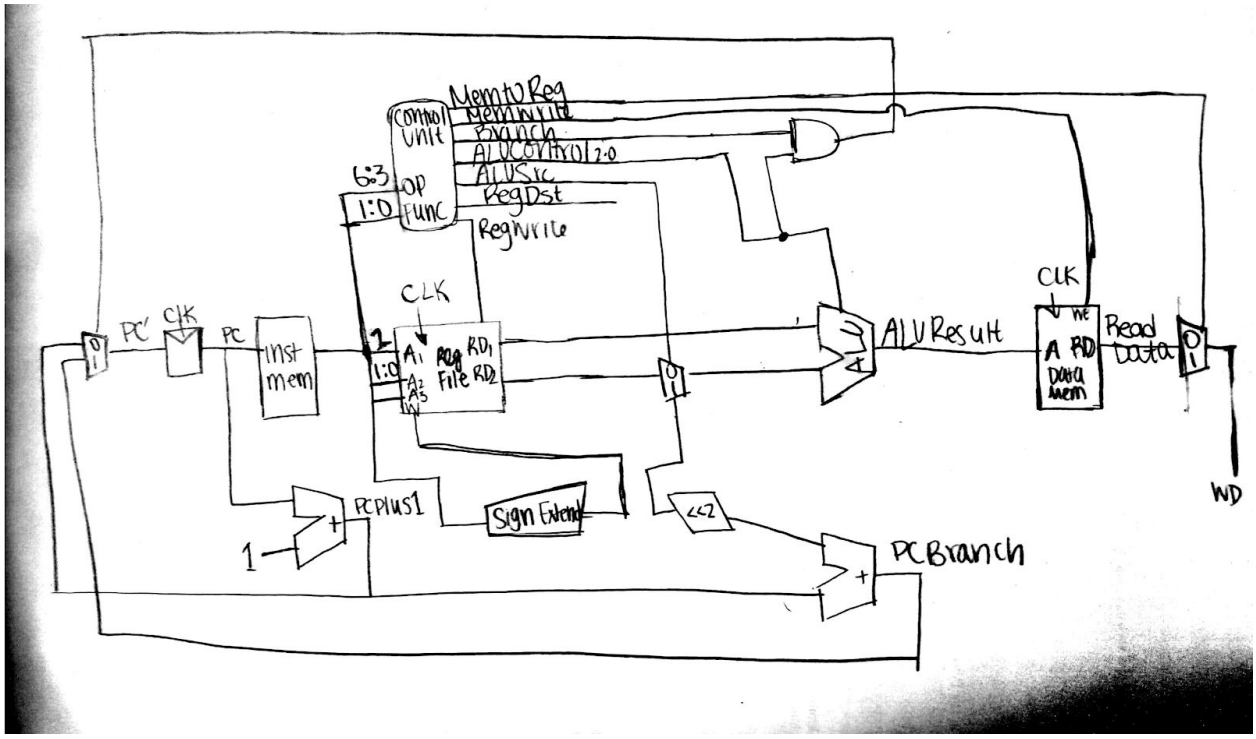
```
        #disassembler(Instructions,Nlines)
        #elif(mode== 3):
        # print( "assembler")
        #assemble(Instructions, Nlines)


        #for line in instr_file: # Reading in the instructions
        #       if (line == "/n" or line[0] == '#'):  #empty lines, comments ignored
        #       line = line.replace("\n," ")
        #       Instructions.append(line)       #Copy all instruction into a list
        #Nline+=1

        #for line in data_file: # Read in data P1_Machine.txt
        #   if(line== "\n" or line[0] =='#'):
#       continue
        #  Memory.append(line)
        #Nlines+=1
        if(mode == 1): #Check whether to use disassembler of assembler or simulator
        #simulator(Instructions,Nsteps,debug_mode,Memory)
        print("assembler")
        elif(mode== 2):
        if (modedis == 1):
        for line in data_file: # Read in data P1_Machine.txt
        if(line== "\n" or line[0] =='#'):
        continue
        Memory.append(line)
        Nlines+=1
        elif(modedis == 2):
        for line in data_file2: # Read in data P1_Machine.txt
        if(line== "\n" or line[0] =='#'):
        continue
        Memory.append(line)
        Nlines+=1
        else:
        print("That is not one of the options")

        disassembler(Memory,Nlines)
        print("disassembler is being done")
        elif(mode== 3):
        #assembler(Instructions,Nlines)
        print("assembler is being done")
        else:
        print("Error. Unrecognized mode. Exiting")
        exit()

        #instr_file.close()
        data_file2.close()
        data_file.close()

if __name__ == "__main__":
        main()
```

Part D) Hardware implementation:

1. ALU schematic. A hierarchical sketch of your Arithmetic Logic Unit which implements whatever computation that your ISA instructions use (See textbook ch 5.2.4).



2. CPU Datapath design. A schematic including your register file, ALU, PC logic, and memory components (see textbook ch 7.3.1).



3. Control logic design. Decoder truth-table indicating how each control signal is generated from an instruction (see textbook ch 7.3.2).

INST	OP	RegWrt	ALUctr	Branch	Memwrt	MemtoReg
Load	1 000	1	1	0	1	1
Store	1 011	0	1	0	0	0
shl	0 001	1	0	0	0	0
sll	0 100	1	0	0	0	0
BezDec	0 010	0	0	1	0	0
xori	0 110	1	0	0	0	0
andi	0 111	1	0	0	0	0
jump	1 010	0	0	1	0	1
sub	1 101	1	0	0	0	0
loadi	0 011	1	1	0	1	1
blt4	0 000	0	0	0	0	0
beq4	1 100	0	0	0	0	0

andi5	1 111	1	0	0	0	0
srl5	1 011	1	0	0	0	0